Tools

Module 5 – Advanced Analytics - Technology and Tools

Module 5: Advanced Analytics - Technology and Tools

Upon completion of this module, you should be able to:

- Perform Analytics on Unstructured data using MapReduce Programming paradigm
- Use Hadoop, HDFS, HIVE,PIG and other products in the Hadoop ecosystem for unstructured data analytics
- Effectively use advanced SQL functions and Greenplum extensions for in-database analytics
- Use MADlib to solve analytics problems in-database

Module 5: Advanced Analytics - Technology and Tools

Lesson 3: In-database Analytics SQL essentials

During this lesson the following topics are covered:

- SQL Essentials
  - SET Operations
  - Online analytical processing (OLAP) features
  - GROUPING SETS, ROLLUP,CUBE
  - GROUPING, GROUP_ID functions
  - Text processing, Pattern matching

Set Operations

Greenplum supports the following set operations as part of a `SELECT` statement:

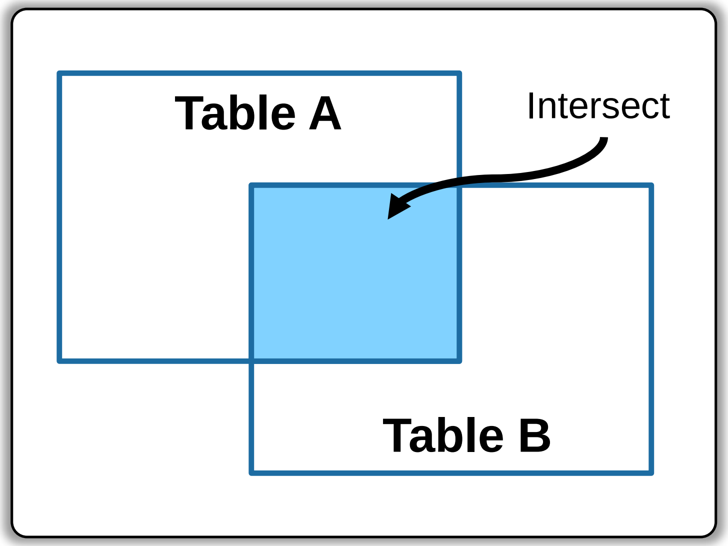
- `INTERSECT` – Returns rows that appear in all answer sets
- `EXCEPT` – Returns rows from the first answer set and excludes those from the second
- `UNION ALL` – Returns a combination of rows from multiple `SELECT` statements with repeating rows
- `UNION` – Returns a combination of rows from multiple `SELECT` statements with no repeating rows

Set Operations – INTERSECT

INTERSECT:

- Returns only the rows that appear in both SQL queries
- Removes duplicate rows

```
SELECT   t.transid,
         c.custname
FROM     facts.transaction t
    JOIN dimensions.customer c
      ON c.customerid = t.customerid

INTERSECT

SELECT t1.transid,
       c1.custname
FROM   facts.transaction t1
  JOIN dimensions.customer c1
    ON c1.customerid = t1.customerid
WHERE  t1.transdate BETWEEN
       ‘2008-01-01’ AND ‘2008-01-21’
```

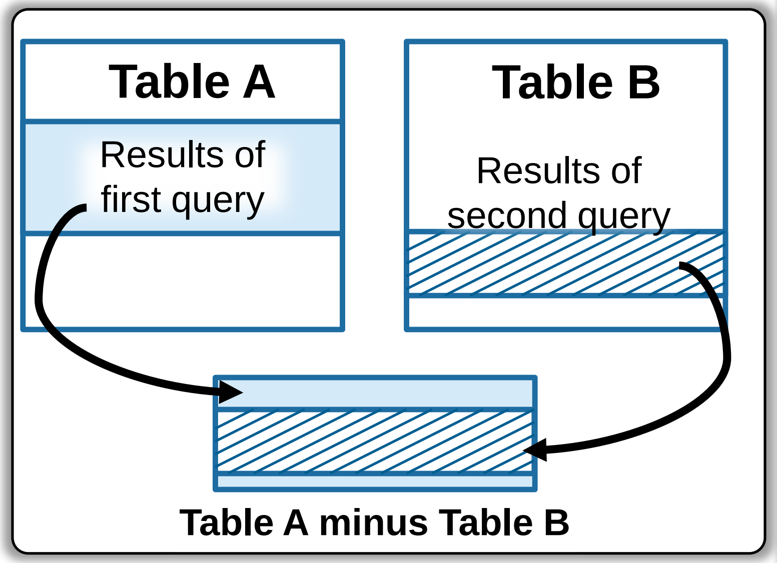


Set Operations – EXCEPT

EXCEPT:

- Returns all rows from the first SELECT statement
- Omits all rows that appear in the second SELECT statement

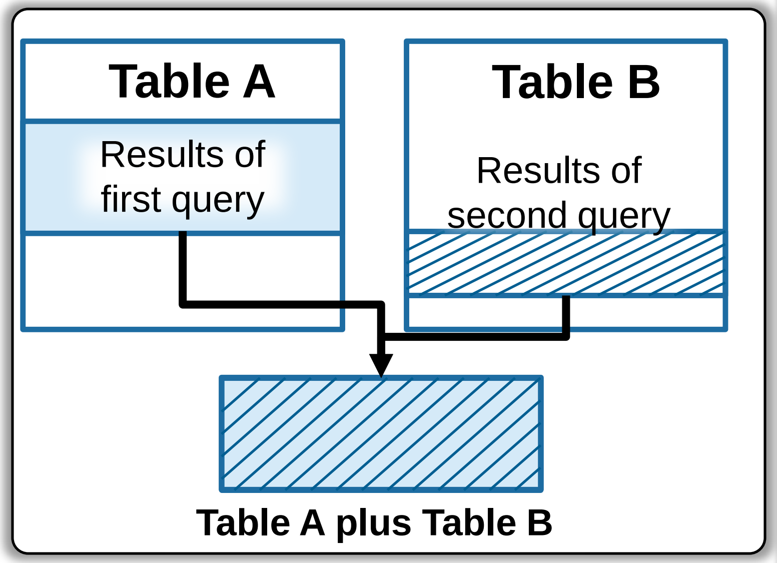
```
SELECT   t.transid,
         c.custname
FROM     facts.transaction t
    JOIN dimensions.customer c
      ON c.customerid = t.customerid

  EXCEPT

SELECT t1.transid,
       c1.custname
FROM   facts.transaction t1
  JOIN dimensions.customer c1
    ON c1.customerid = t1.customerid
WHERE  t1.transdate BETWEEN
       ‘2008-01-01’ AND ‘2008-01-21’
```



Set Operations – UNION ALL

UNION ALL:

- Combines rows from the first query with rows from the second query
- Does not remove duplicates rows



```
SELECT t.transid,
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       ‘2008-01-01’ AND ‘2008-05-17’

  UNION ALL

SELECT t1.transid,
       c1.custname
FROM   facts.transaction t1
  JOIN dimensions.customer c1
    ON c1.customerid = t1.customerid
WHERE  t1.transdate BETWEEN
       ‘2008-01-01’ AND ‘2008-01-21’
```

Set Operations – UNION

UNION:

- Combines rows from the first query with rows from the second query
- Removes duplicates or repeating rows

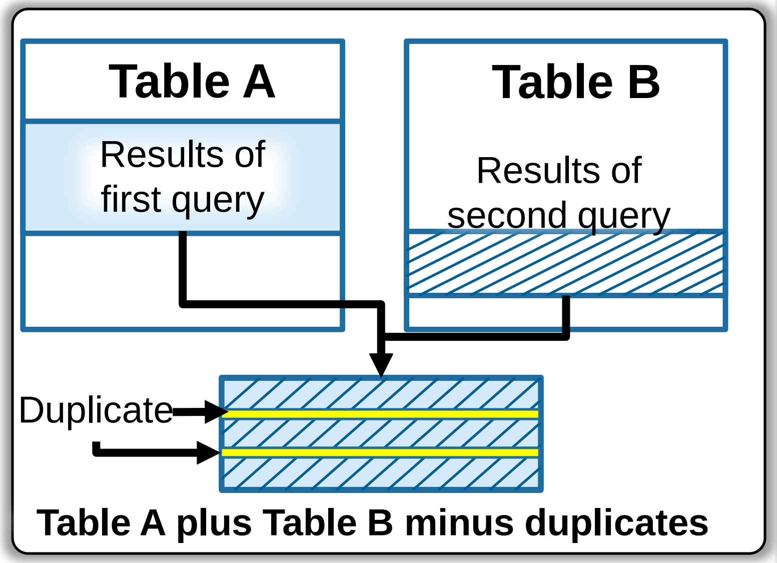
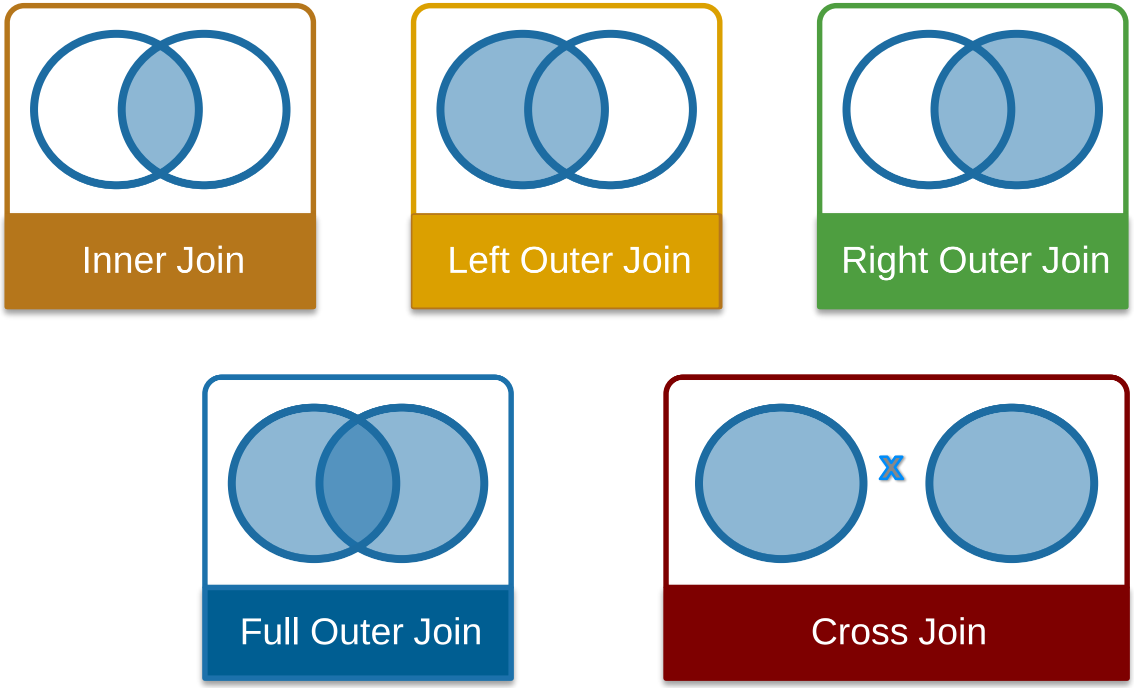


Table A plus Table B minus duplicates

```
SELECT t.transid,
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       ‘2008-01-01’ AND ‘2008-05-17’

  UNION

SELECT t1.transid,
       c1.custname
FROM   facts.transaction t1
  JOIN dimensions.customer c1
    ON c1.customerid = t1.customerid
WHERE  t1.transdate BETWEEN
       ‘2008-01-01’ AND ‘2008-01-21’
```

SET Operations

- Types of Join
  - Inner
  - Left outer
  - Right outer
  - Full outer
  - Cross



Left Outer Join

- Correlated sub-queries do not run efficiently in Greenplum though support has been introduced in Version 4.2

  ```
  SELECT * FROM transaction t
  WHERE NOT EXISTS (
      SELECT 1 FROM customer c
      WHERE c.customerid = t.customerid)
  ```

- Use LEFT OUTER JOIN

  ```
  SELECT t.* FROM transaction t
  LEFT OUTER JOIN
      customer c ON t.customerid=c.customerid
  WHERE c.customerid IS NULL
  ```

Sub-query vs. Inner Join

- IN clause is fully supported …

  ```
  SELECT *
  FROM transaction t
  WHERE t.customerid IN
    (SELECT customerid FROM customer)
  ```

- However, generally better idea as long as c.customerid is unique:

  ```
  SELECT t.*
  FROM transaction t
  INNER JOIN customer c
  ON c.customerid = t.customerid
  ```

Greenplum SQL OLAP Grouping Extensions

Greenplum supports the following grouping extensions:

- Standard `GROUP BY`
- `ROLLUP`
- `GROUPING SETS`
- `CUBE`
- `grouping(column [, ...])` function
- `group_id()` function

Standard GROUP BY Example

GROUP BY:

- Group results based on one or more specified columns
- Is used with aggregate statements

The following example summarizes product sales by vendor:

```
SELECT pn, vn, sum(prc*qty)
FROM sale
GROUP BY pn, vn
ORDER BY 1,2,3;
```

```
pn | vn | sum
-----+----+--------
100 | 20 | 0
100 | 40 | 2640000
200 | 10 | 0
200 | 40 | 0
300 | 30 | 0
400 | 50 | 0
500 | 30 | 120
600 | 30 | 60
700 | 40 | 1
800 | 40 | 1
(10 rows)
```

Standard GROUP BY Example with UNION ALL

This example extends the previous example by adding sub-totals and a grand total :

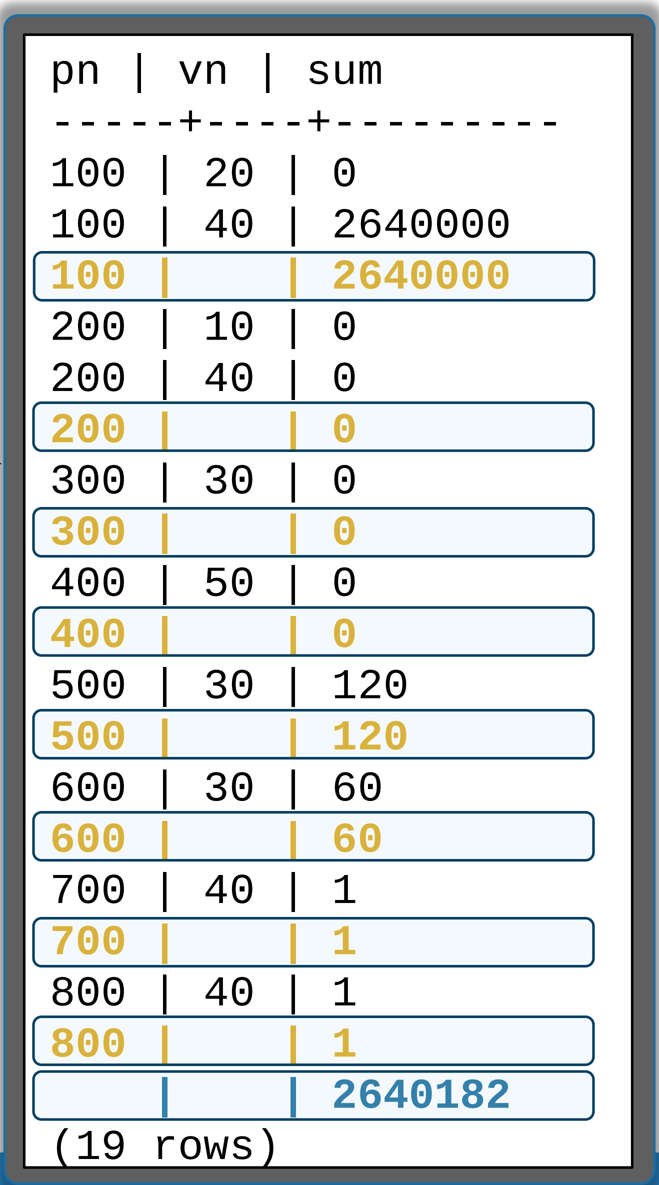
```
SELECT pn, vn, sum(prc*qty)
FROM sale
GROUP BY pn, vn
UNION ALL
SELECT pn, null, sum(prc*qty)
FROM sale
GROUP BY pn
UNION ALL
SELECT null, null,
sum(prc*qty)
FROM SALE
ORDER BY 1,2,3;
```

```
pn  | vn  | sum
-----+----+---------
100 | 20  | 0
100 | 40  | 2640000
100 |     | 2640000
200 | 10  | 0
200 | 40  | 0
200 |     | 0
300 | 30  | 0
300 |     | 0
400 | 50  | 0
400 |     | 0
500 | 30  | 120
500 |     | 120
600 | 30  | 60
600 |     | 60
700 | 40  | 1
700 |     | 1
800 | 40  | 1
800 |     | 1
    |     | 2640182
(19 rows)
```

ROLLUP Example

The following example meets the requirement where the sub-total and grand totals are to be included:

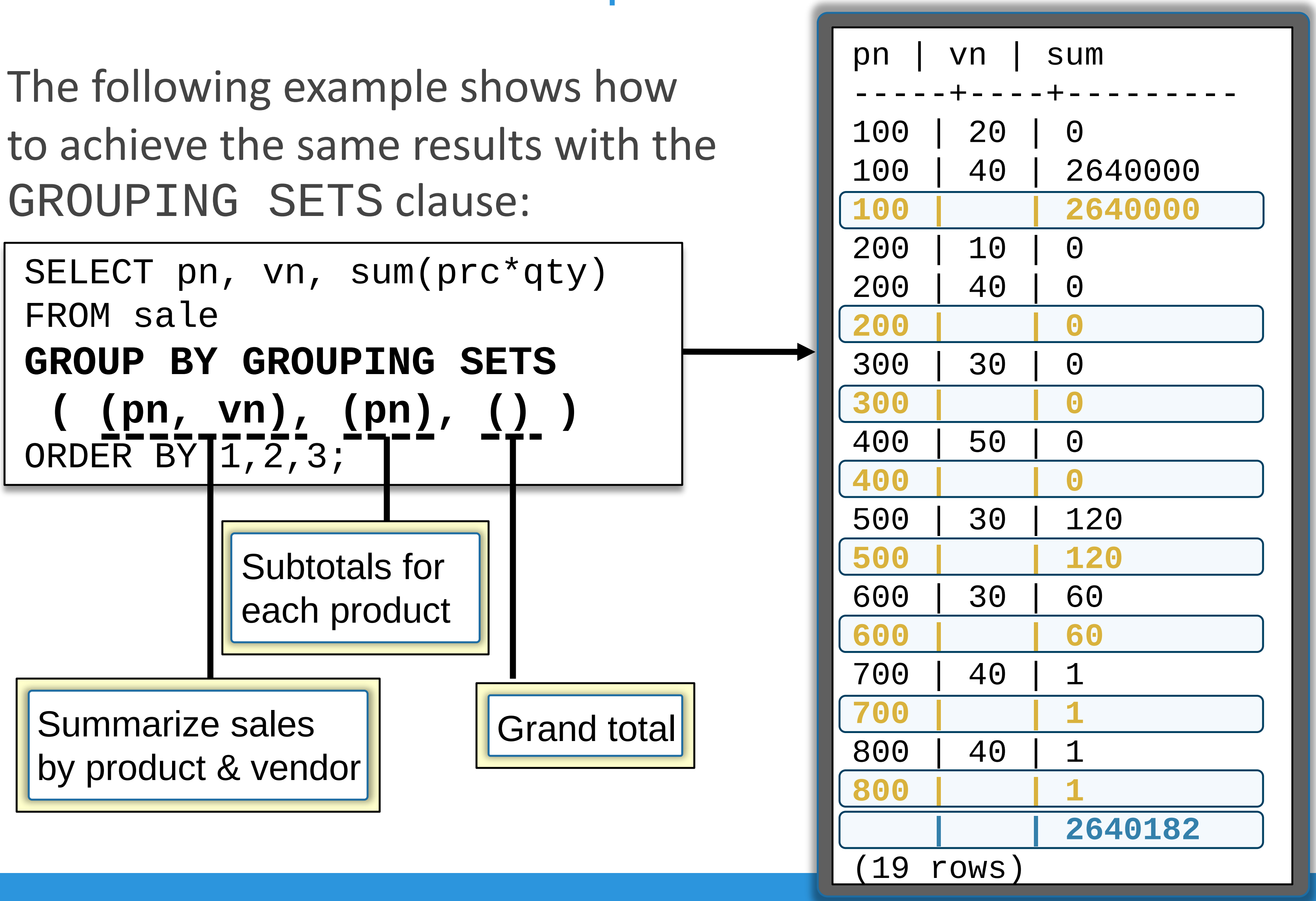
```
SELECT pn, vn, sum(prc*qty)
FROM sale
GROUP BY ROLLUP(pn, vn)
ORDER BY 1,2,3;
```



```
pn | vn | sum
-----+----+----------
100 | 20 | 0
100 | 40 | 2640000
100 |    | 2640000
200 | 10 | 0
200 | 40 | 0
200 |    | 0
300 | 30 | 0
300 |    | 0
400 | 50 | 0
400 |    | 0
500 | 30 | 120
500 |    | 120
600 | 30 | 60
600 |    | 60
700 | 40 | 1
700 |    | 1
800 | 40 | 1
800 |    | 1
    |    | 2640182
(19 rows)
```

GROUPING SETS Example

The following example shows how to achieve the same results with the GROUPING SETS clause:



```
SELECT pn, vn, sum(prc*qty)
FROM sale
GROUP BY GROUPING SETS
 ( (pn, vn), (pn), () )
ORDER BY 1,2,3;
```

Summarize sales by product & vendor

Subtotals for each product

Grand total

```
pn | vn | sum
-----+-----+----------
100 | 20 | 0
100 | 40 | 2640000
100 |    | 2640000
200 | 10 | 0
200 | 40 | 0
200 |    | 0
300 | 30 | 0
300 |    | 0
400 | 50 | 0
400 |    | 0
500 | 30 | 120
500 |    | 120
600 | 30 | 60
600 |    | 60
700 | 40 | 1
700 |    | 1
800 | 40 | 1
800 |    | 1
    |    | 2640182
(19 rows)
```

CUBE Example

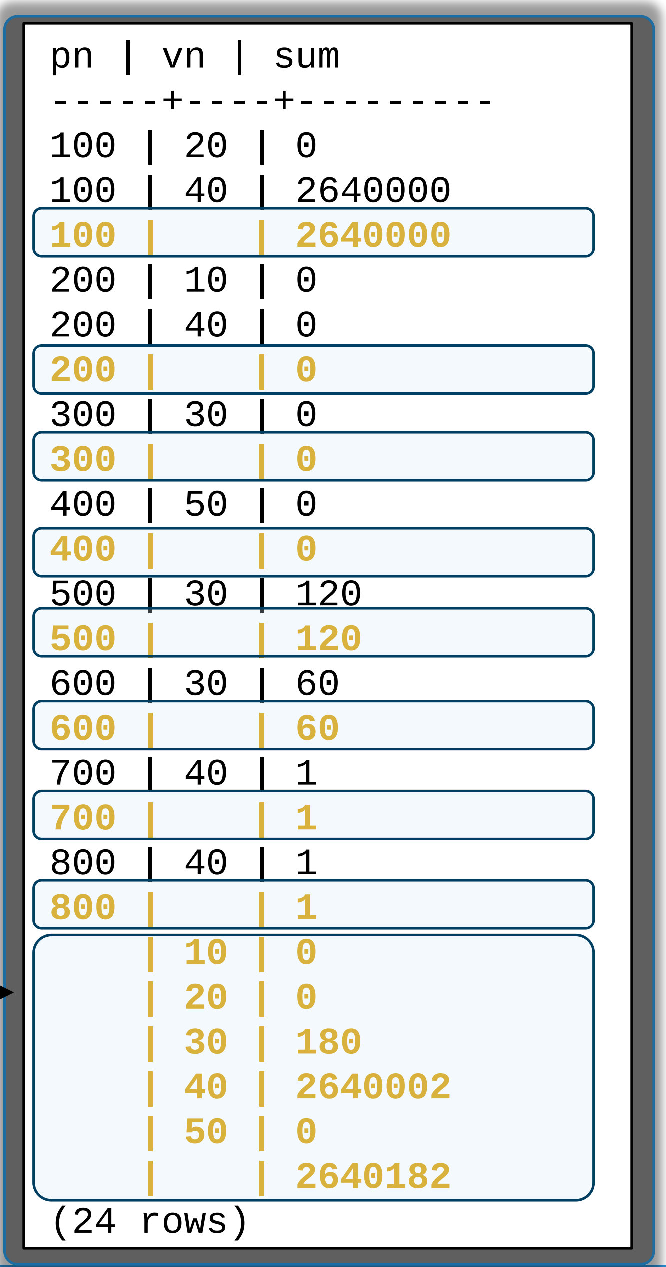
CUBE creates subtotals for all possible combinations of grouping columns.

The following example

```
SELECT pn, vn, sum(prc*qty)
FROM sale
GROUP BY CUBE(pn, vn)
ORDER BY 1,2,3;
```

is the same as

```
SELECT pn, vn, sum(prc*qty)
FROM sale
GROUP BY GROUPING SETS
     ( (pn, vn), (pn),
(vn), () )
ORDER BY 1,2,3;
```



```
pn  | vn | sum
-----+----+---------
100 | 20 | 0
100 | 40 | 2640000
100 |    | 2640000
200 | 10 | 0
200 | 40 | 0
200 |    | 0
300 | 30 | 0
300 |    | 0
400 | 50 | 0
400 |    | 0
500 | 30 | 120
500 |    | 120
600 | 30 | 60
600 |    | 60
700 | 40 | 1
700 |    | 1
800 | 40 | 1
800 |    | 1
    | 10 | 0
    | 20 | 0
    | 30 | 180
    | 40 | 2640002
    | 50 | 0
    |    | 2640182
(24 rows)
```

GROUPING Function Example

Grouping distinguishes NULL from summary markers.

```
store   | customer   | product   | price
-------+----------+---------+-------
s2      | c1         | p1        | 90
s2      | c1         | p2        | 50
s2      |            | p1        | 44
s1      | c2         | p2        | 70
s1      | c3         | p1        | 40
(5 rows)
```

```
SELECT * FROM dsales_null;
```

```
SELECT
store,customer,product,
sum(price),
    grouping(customer)
FROM dsales_null
GROUP BY
ROLLUP(store,customer,
    product);
```

```
store   | customer | product  | sum | grouping
-------+----------+---------+----+----------
s1      | c2       | p2       | 70  | 0
s1      | c2       |          | 70  | 0
s1      | c3       | p1       | 40  | 0
s1      | c3       |          | 40  | 0
s1      |          |          | 110 | 1
s2      | c1       | p1       | 90  | 0
s2      | c1       | p2       | 50  | 0
s2      | c1       |          | 140 | 0
s2      |          | p1       | 44  | 0
s2      |          |          | 44  | 0
s2      |          |          | 184 | 1
        |          |          | 294 | 1
(12 rows)
```

GROUP_ID Function

GROUP_ID:

- Returns 0 for each output row in a unique grouping set
- Assigns a serial number >0 to each duplicate grouping set found
- Can be used to filter output rows of duplicate grouping sets, such as in the following example:

```
SELECT a, b, c, sum(p*q), group_id()
FROM sales
GROUP BY ROLLUP(a,b), CUBE(b,c)
HAVING group_id()<1
ORDER BY a,b,c;
```

In-database Text Analysis

- SQL features for
  - Text handling functions
  - Pattern matching with regular expressions
- Example Use-cases
  - Filter emails with spam tag in subject
  - Extract domains from a URL
  - Extract all URLs from a HTML file
  - Check for Syntactically correct email addresses
  - Convert 10 digits into format “(123) 456-7890”

Pattern Matching - Regular Expressions (Regex)

Regular Expression match Operators

Operator	Description	Example
~	**Case sensitive substring**	**‘Greenplum’ ~ ‘^Green’**
~*	**Case-insensitive substring**	**‘Greenplum’ ~*’ee+’**

SQL Functions

- substring(*string,* from,*pattern* [for *escape*])
- regexp_matches(*string*, *pattern*, [*flags*])
- regexp_replace(*string*, *pattern*, *repl*, [*flags*])
- regexp_split_to_{array|table}

Regular Expression Quantifiers

- Quantifier

Expression	Matches
.	Arbitrary character
^ And $	Virtual characters for beginning and end
*	Preceding item zero or more times
+	Preceding item one or more times
?	Preceding item is optional
{*n*}	Preceding item *n* times
a|b	Item *a* or *b*
…	…

Check Your Knowledge

Your Thoughts?

1. How would you use GROUPING SETS to produce the same results as the following GROUP BY CUBE?

   SELECT state, productID, SUM(volume) FROM sales GROUP BY CUBE (state, productID) ORDER BY state, productID

2. How would you show the sub-totals for each week, for each state, and for each product? (No other totals or grand totals are required.) Suppose the table structure is

   TABLE sales (productID VARCHAR, state CHAR(2), week DATE, volume INT)

3. Discuss the utility of grouping and group_id functions

Check Your Knowledge (Continued)

Your Thoughts?

4. Give regular expressions for the following:
   - A regex that, given a URL, captures the domain name
   - A regex that captures PostgreSQL Dollar-quoted String literals
     Examples:
     - $test$This is a string$test$

Module 5: Advanced Analytics - Technology and Tools

Lesson 3: Summary

During this lesson the following SQL Essentials topics were covered:

- Online analytical processing (OLAP) features
- GROUPING SETS, ROLLUP,CUBE
- GROUPING, GROUP_ID functions
- Text processing, Pattern matching

Module 5: Advanced Analytics - Technology and Tools

Lesson 4: Advanced SQL and MADlib

During this lesson the following topics are covered:

Advanced SQL and MADlib:

- Window functions
- User defined functions and aggregates
- Ordered Aggregates
- MADlib

Window Functions

- About Window Functions
  - Returns a value per row, unlike aggregate functions
  - Has its results interpreted in terms of the current row and its corresponding window partition or frame
  - Is characterized by the use of the OVER clause
  - Defines the window partitions, or groups of rows to apply the function
  - Defines ordering of data within a window
  - Defines the positional or logical framing of a row with respect to its window

Window Functions (Continued)

- About Window Functions
  - Returns a value per row, unlike aggregate functions
  - Has its results interpreted in terms of the current row and its corresponding window partition or frame
  - Is characterized by the use of the OVER clause
  - Defines the window partitions, or groups of rows to apply the function
  - Defines ordering of data within a window
  - Defines the positional or logical framing of a row with respect to its window

Defining Window Specifications (OVER Clause)

When defining the window function:

  - Include an `OVER( )` clause
  - Specify the window of data to which the function applies

- Define:
  - Window partitions, using the `PARTITION BY` clause
  - Ordering within a window partition, using the `ORDER BY` clause
  - Framing within a window partition, using `ROWS` and `RANGE` clauses
  - The ORDER BY clause also defines a frame of unbounded preceding to current in the partition

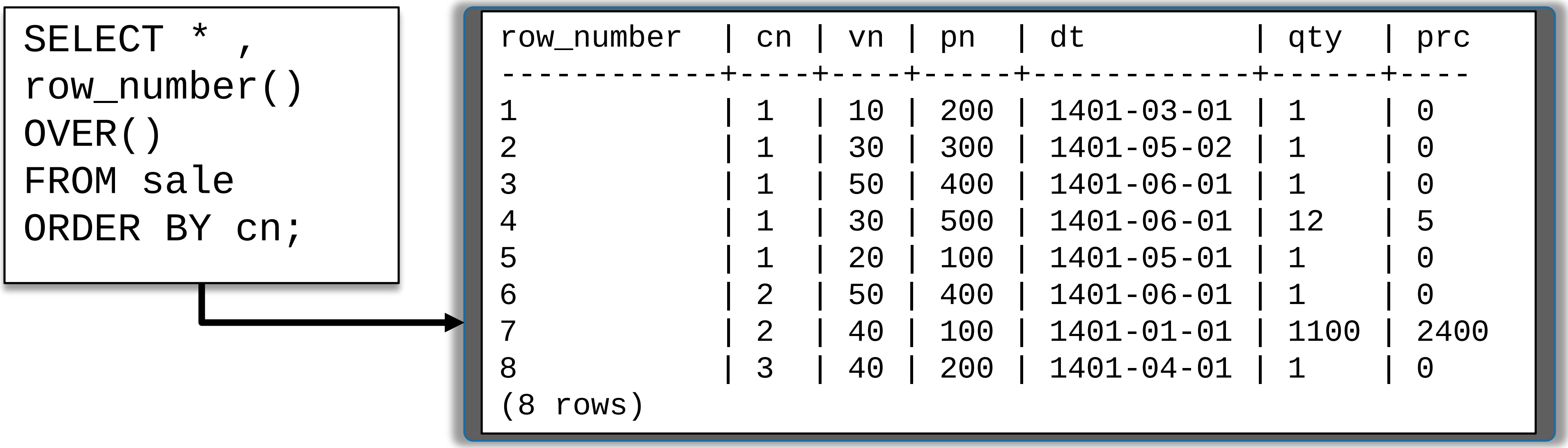
About the PARTITION BY Clause

The PARTITION BY clause:

- Can be used by all window functions
- Organizes result sets into groupings based on unique values
- Allows the function to be applied to each partition independently

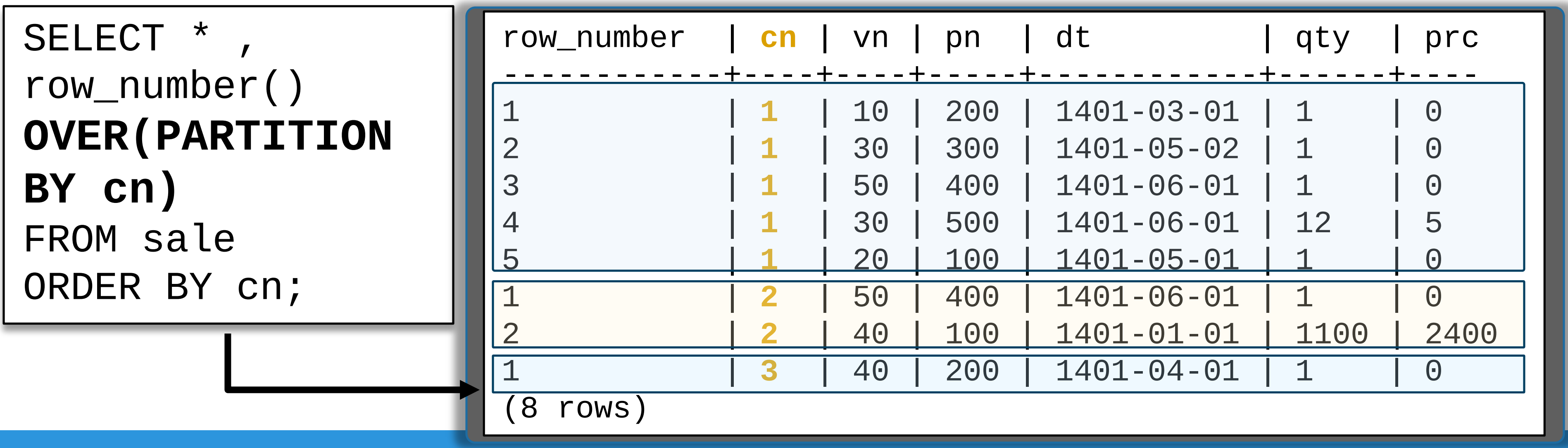
Note: If the PARTITION BY clause is omitted, the entire result set is treated as a single window partition.

Window Partition Example



```
SELECT * ,
row_number()
OVER()
FROM sale
ORDER BY cn;
```

```
row_number   | cn | vn | pn  | dt          | qty  | prc
-------------+----+----+-----+-------------+------+----
1            | 1  | 10 | 200 | 1401-03-01  | 1    | 0
2            | 1  | 30 | 300 | 1401-05-02  | 1    | 0
3            | 1  | 50 | 400 | 1401-06-01  | 1    | 0
4            | 1  | 30 | 500 | 1401-06-01  | 12   | 5
5            | 1  | 20 | 100 | 1401-05-01  | 1    | 0
6            | 2  | 50 | 400 | 1401-06-01  | 1    | 0
7            | 2  | 40 | 100 | 1401-01-01  | 1100 | 2400
8            | 3  | 40 | 200 | 1401-04-01  | 1    | 0
(8 rows)
```



```
SELECT * ,
row_number()
OVER(PARTITION
BY cn)
FROM sale
ORDER BY cn;
```

```
row_number   | cn | vn | pn  | dt          | qty  | prc
-------------+----+----+-----+-------------+------+----
1            | 1  | 10 | 200 | 1401-03-01  | 1    | 0
2            | 1  | 30 | 300 | 1401-05-02  | 1    | 0
3            | 1  | 50 | 400 | 1401-06-01  | 1    | 0
4            | 1  | 30 | 500 | 1401-06-01  | 12   | 5
5            | 1  | 20 | 100 | 1401-05-01  | 1    | 0
1            | 2  | 50 | 400 | 1401-06-01  | 1    | 0
2            | 2  | 40 | 100 | 1401-01-01  | 1100 | 2400
1            | 3  | 40 | 200 | 1401-04-01  | 1    | 0
(8 rows)
```

RANK and ORDER BY

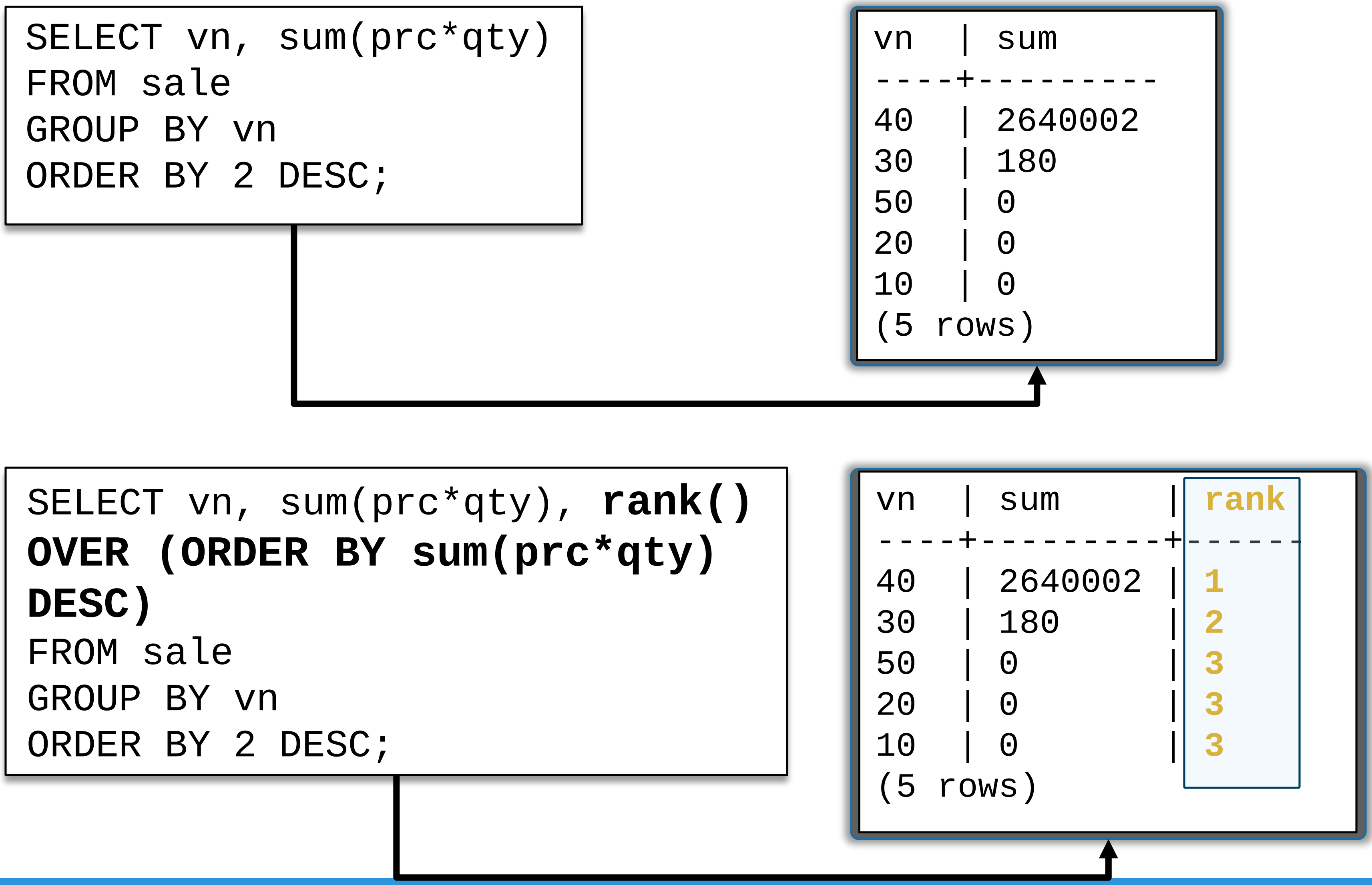
The ORDER BY clause:

- Can always be used by window functions
- Is required by some window functions such as RANK
- Specifies ordering within a window partition

The RANK built-in function:

- Calculates the rank of a row
- Gives rows with equal values for the specified criteria the same rank

Using the OVER (ORDER BY...) Clause



```
SELECT vn, sum(prc*qty)
FROM sale
GROUP BY vn
ORDER BY 2 DESC;
```

```
vn   | sum
----+---------
40   | 2640002
30   | 180
50   | 0
20   | 0
10   | 0
(5 rows)
```

```
SELECT vn, sum(prc*qty), rank()
OVER (ORDER BY sum(prc*qty)
DESC)
FROM sale
GROUP BY vn
ORDER BY 2 DESC;
```

```
vn   | sum        | rank
----+----------+------
40   | 2640002  | 1
30   | 180      | 2
50   | 0        | 3
20   | 0        | 3
10   | 0        | 3
(5 rows)
```

Designating a Sliding (Moving) Window

A moving window:

- Defines a set or rows in a window partition
- Allows you to define the first row and last row
- Uses the current row as the reference point
- Can be expressed in rows with the ROWS clause
- Can be expressed as a range with the RANGE clause

Designating a Sliding (Moving) Window (Continued)

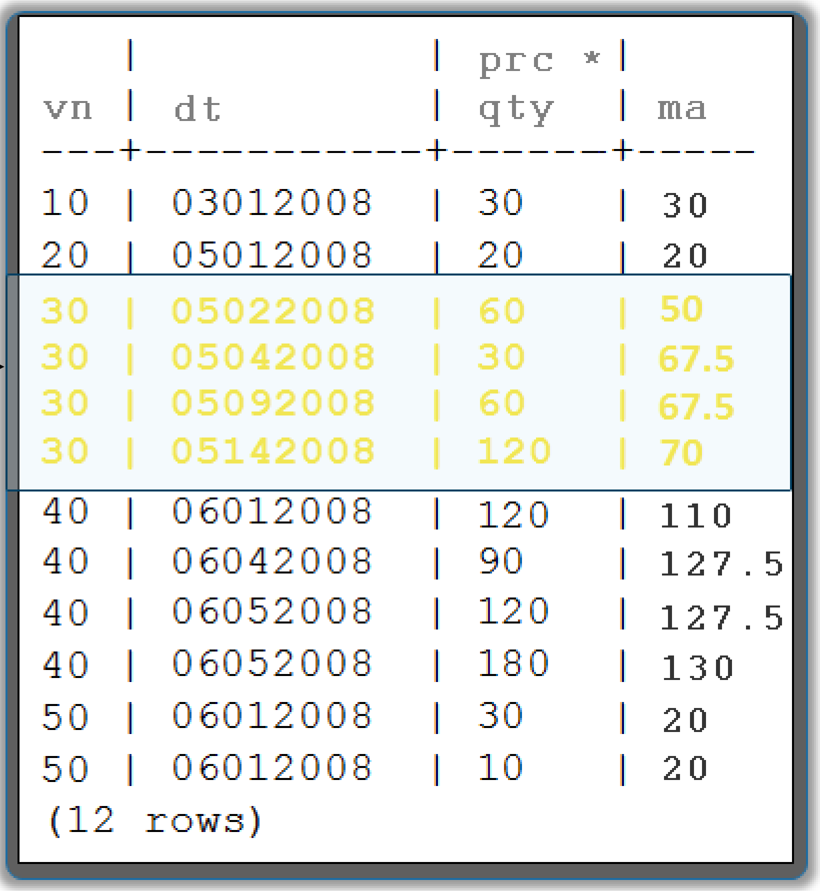
A moving window:

- Defines a set or rows in a window partition
- Allows you to define the first row and last row
- Uses the current row as the reference point
- Can be expressed in rows with the ROWS clause
- Can be expressed as a range with the RANGE clause

Window Framing Example

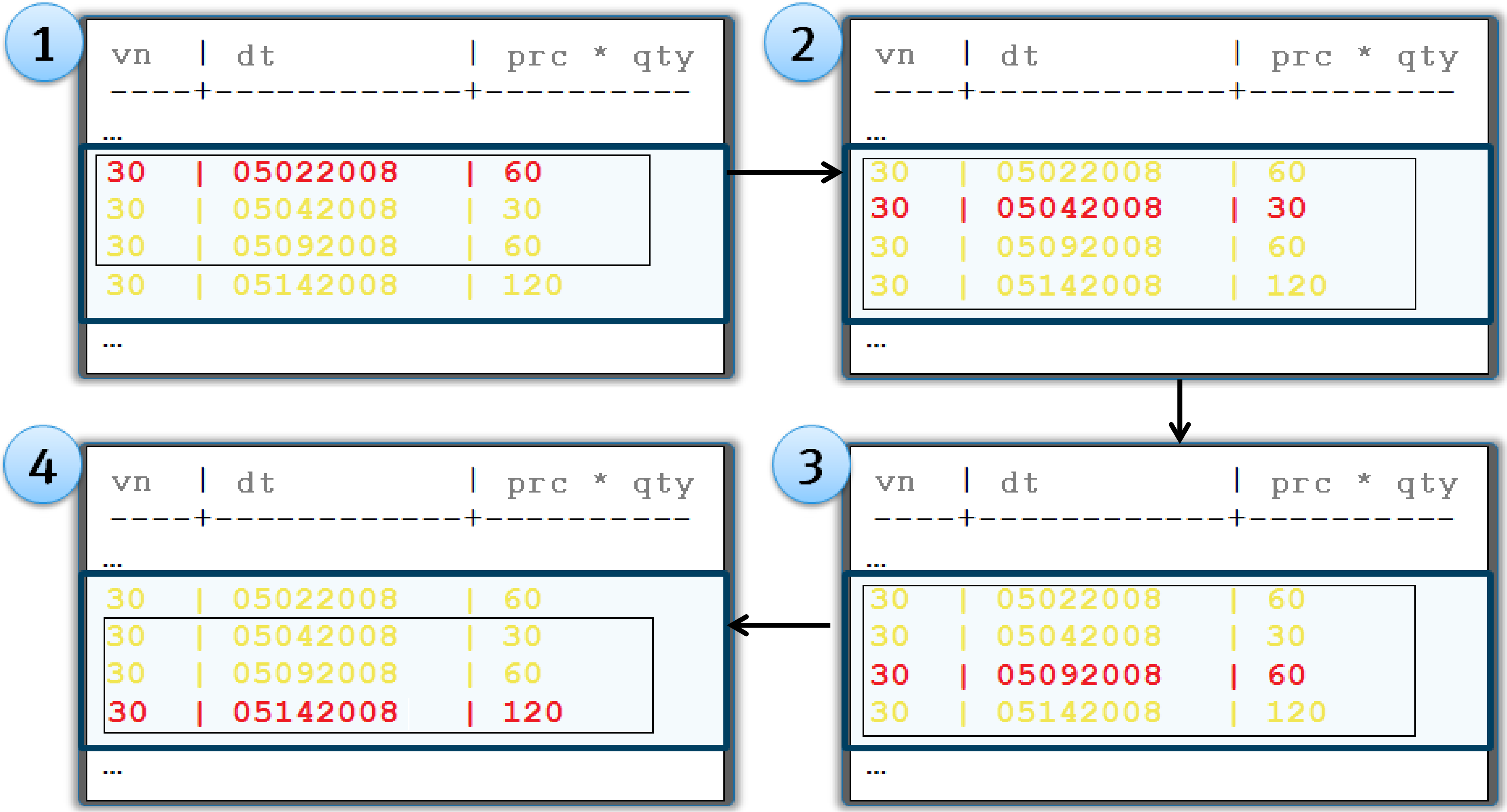
A rolling window moves through a partition of data, one row at a time.

```
  SELECT vn, dt, prc * qty
      ma = AVG(prc*qty)
      OVER (PARTITION BY vn
          ORDER BY dt
          ROWS BETWEEN
          2 PRECEDING AND
          2 FOLLOWING)
  FROM sale;
```



```
    |            | prc * |
vn  | dt         | qty   | ma
---+-----------+-------+-----
10 | 03012008  | 30    | 30
20 | 05012008  | 20    | 20
30 | 05022008  | 60    | 50
30 | 05042008  | 30    | 67.5
30 | 05092008  | 60    | 67.5
30 | 05142008  | 120   | 70
40 | 06012008  | 120   | 110
40 | 06042008  | 90    | 127.5
40 | 06052008  | 120   | 127.5
40 | 06052008  | 180   | 130
50 | 06012008  | 30    | 20
50 | 06012008  | 10    | 20
(12 rows)
```

Window Framing Example (Continued)



1

```
vn  | dt          | prc * qty
----+-------------+----------
...
30  | 05022008    | 60
30  | 05042008    | 30
30  | 05092008    | 60
30  | 05142008    | 120
...
```

2

```
vn  | dt          | prc * qty
----+-------------+----------
...
30  | 05022008    | 60
30  | 05042008    | 30
30  | 05092008    | 60
30  | 05142008    | 120
...
```

3

```
vn  | dt          | prc * qty
----+-------------+----------
...
30  | 05022008    | 60
30  | 05042008    | 30
30  | 05092008    | 60
30  | 05142008    | 120
...
```

4

```
vn  | dt          | prc * qty
----+-------------+----------
...
30  | 05022008    | 60
30  | 05042008    | 30
30  | 05092008    | 60
30  | 05142008    | 120
...
```

General Syntax of Window Function

A moving window:

- Is defined as part of a window with the ORDER BY clause as follows:

```
WINDOW window_name AS (window_specification)
```

where window_specification can be:

```
[window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [, ...]
[{RANGE | ROWS}
{ UNBOUNDED PRECEDING
| expression PRECEDING
| CURRENT ROW
| BETWEEN window_frame_bound AND window_frame_bound }]]
```

Built-In Window Functions

Built-In Function	Description
`dist()`	Calculates the cumulative distribution of a value in a group of values. Rows with equal values always evaluate to the same cumulative distribution value.
`dense_rank()`	Computes the rank of a row in an ordered group of rows without skipping rank values. Rows with equal values are given the same rank value.
first_value(*expr*)	Returns the first value in an ordered set of values.
lag(expr [,*offset*] [,default])	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position. If offset is not specified, the default offset is 1. default sets the value that is returned if the offset goes beyond the scope of the window. If default is not specified, the default value is null.

Note: Any aggregate function used with the OVER clause can also be used as a window function.

Built-In Window Functions (Continued)

Built-In Function	Description
`last_value(`*`expr`*`)`	Returns the last value in an ordered set of values.
`lead()`	Provides access to more than one row of the same table without doing a self join. Given a series of rows returned from a query and a position of the cursor, LEAD provides access to a row at a given physical offset after that position. If offset is not specified, the default offset is 1. default sets the value that is returned if the offset goes beyond the scope of the window. If default is not specified, the default value is null.
`ntile(expr)`	Divides an ordered dataset into a number of buckets (as defined by expr) and assigns a bucket number to each row.
`percent_rank()`	Calculates the rank of a hypothetical row R minus 1, divided by 1 less than the number of rows being evaluated (within a window partition).
`row_number()`	Assigns a unique number to each row to which it is applied (either each row in a window partition or each row of the query).

Check Your Knowledge

Your Thoughts?

- Describe how this code will work:

```
SELECT dt, region, revenue,
  count(*) OVER (twdw) AS moving_count,
  avg(revenue) OVER (twdw) AS moving_average
FROM moving_average_data mad
WINDOW twdw AS (PARTITION BY region
    ORDER BY dt RANGE BETWEEN
    '7 days'::interval PRECEDING AND
    '0 days'::interval FOLLOWING)
ORDER BY region, dt
```

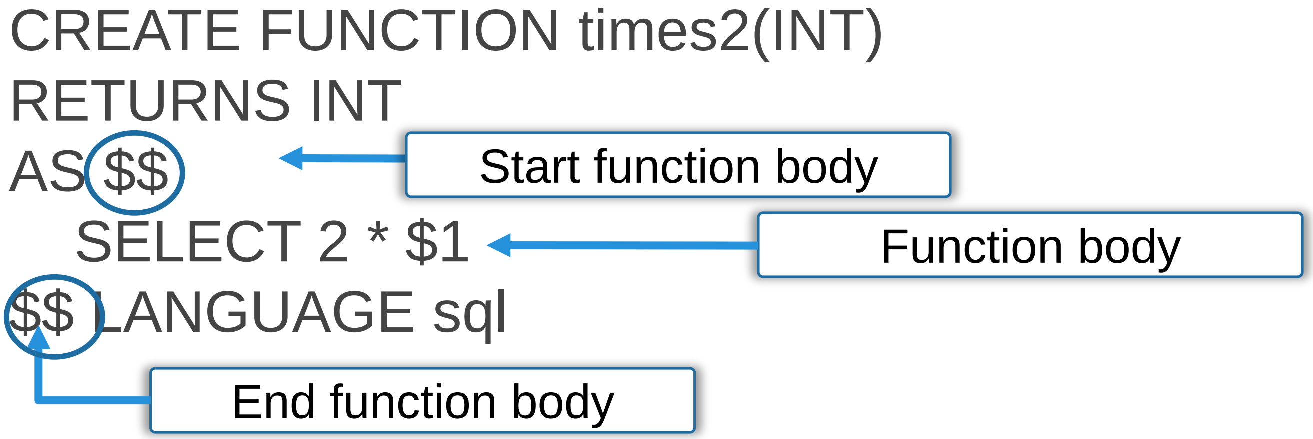
User Defined Functions and Aggregates

Greenplum supports several function types, including:

- Query language functions where the functions are written in SQL
- Procedural language functions where the functions are written in:
    - PL/pgSQL
    - PL/TcL
    - Perl
    - Python
    - R
- Internal functions
- C-language functions
- Use Case examples:
    - Second largest element in a column?
    - Online auction: Who is the second highest bidder?

Anatomy of a User-Defined Function

- Example:



```
CREATE FUNCTION times2(INT)
RETURNS INT
AS $$
    SELECT 2 * $1
$$ LANGUAGE sql
```

Start function body

Function body

End function body

```
SELECT times2(1);
 times2
--------
      2
(1 row)
```

User-Defined Aggregates

- Perform a single table scan
- Example: Second largest number
  - Keep a state: maximum 2 numbers
  - New number can displace the smaller one in the state
  - Greenplum extension: Merge two states
- Example :Create a sum of cubes aggregate:

```
CREATE FUNCTION scube_accum(numeric, numeric) RETURNS numeric
AS 'select $1 + $2 * $2 * $2'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;

CREATE AGGREGATE scube(numeric) (
SFUNC = scube_accum,
STYPE = numeric,
INITCOND = 0 );
```

Ordered Aggregates

- Output of aggregates may depend on order
  - Example:
    SELECT array_agg(letter) FROM alphabet
  - SQL does not guarantee a particular order
  - Output could be {a,b,c} or {b,c,d} or ... depending on query optimizer, distribution of data, ...
- Sample Use Case:
    - Maximum value of discrete derivative? For example:
      Largest single-day stock increase during last year?
- Greenplum 4.1 introduces ordered aggregates:
  - SELECT array_agg(*column* ORDER BY *expression* [ASC|DESC]) FROM *table*
- Median can be implemented using an ordered call of array_agg()
  - This will be covered in the Lab

MADlib: Definition

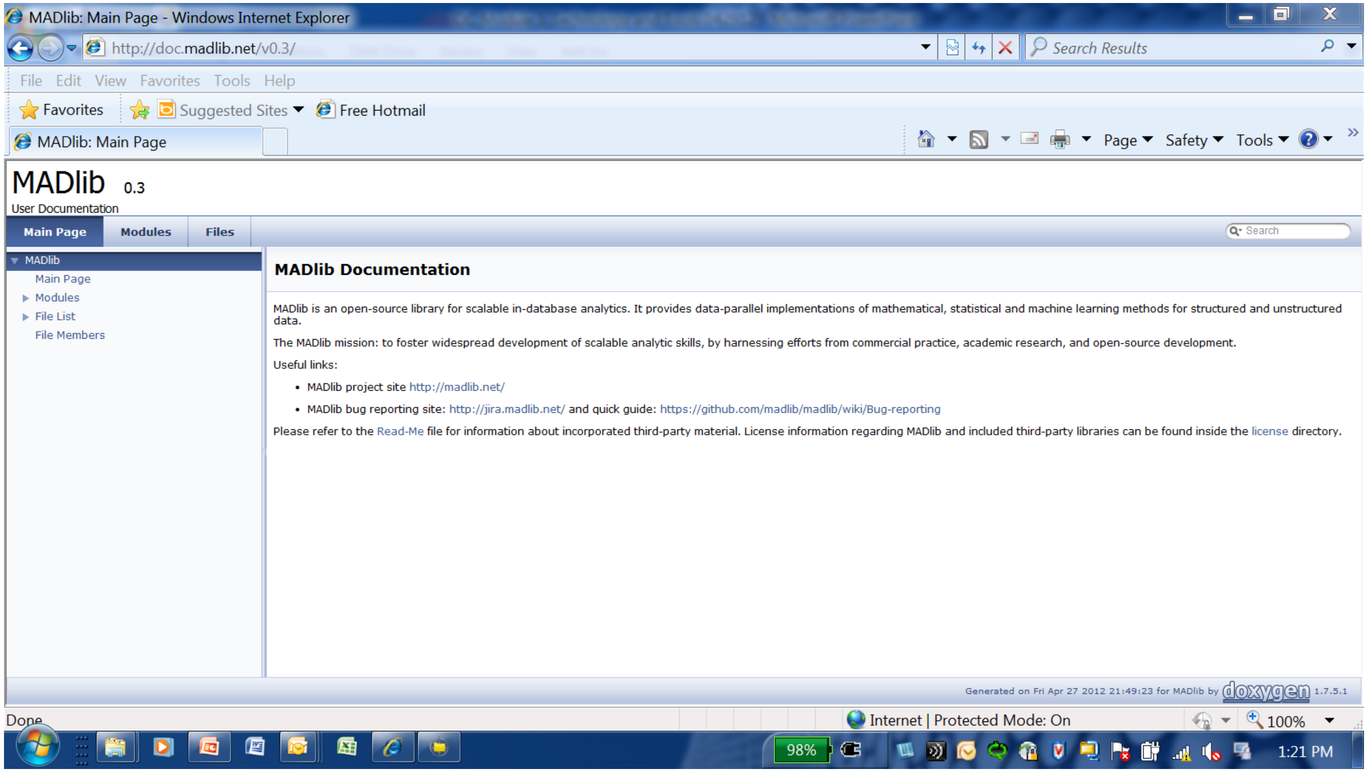
- **MAD** stands for:

- **lib** stands for library of:
  - advanced (mathematical, statistical, machine learning)
  - parallel & scalable
  - in-database functions

- **Mission:** to foster widespread development of scalable analytic skills, by harnessing efforts from commercial practice, academic research, and open-source development.

MADlib: Getting Help…

- Check out the **user guide** with examples at: *http://doc.madlib.net*



MADlib: Main Page - Windows Internet Explorer
http://doc.madlib.net/v0.3/
Search Results
File Edit View Favorites Tools Help
Favorites
Suggested Sites
Free Hotmail
MADlib: Main Page
Page Safety Tools
Main Page
Modules
Files
Search
Done
Internet | Protected Mode: On
100%
98%
1:21 PM

MADlib 0.3

User Documentation

MADlib
- Main Page
- Modules
- File List
- File Members

MADlib Documentation

MADlib is an open-source library for scalable in-database analytics. It provides data-parallel implementations of mathematical, statistical and machine learning methods for structured and unstructured data.

The MADlib mission: to foster widespread development of scalable analytic skills, by harnessing efforts from commercial practice, academic research, and open-source development.

Useful links:

- MADlib project site http://madlib.net/
- MADlib bug reporting site: http://jira.madlib.net/ and quick guide: https://github.com/madlib/madlib/wiki/Bug-reporting

Please refer to the Read-Me file for information about incorporated third-party material. License information regarding MADlib and included third-party libraries can be found inside the license directory.

Generated on Fri Apr 27 2012 21:49:23 for MADlib by doxygen 1.7.5.1

- Need more help?
  Try: http://groups.google.com/group/madlib-user-forum

Greenplum In-database Analytical Functions

Descriptive Statistics	Modeling
Quantile	**Association Rule Mining**
Profile	**K-Means Clustering**
CountMin (Cormode-Muthukrishnan) Sketch-based Estimator	**Naïve Bayes Classification**
FM (Flajolet-Martin) Sketch-based Estimator	**Linear Regression**
MFV (Most Frequent Values) Sketch-based Estimator	**Logistic Regression**
Frequency	**Support Vector Machines**
Histogram	SVD Matrix Factorization
Bar Chart	**Decision Trees/CART**
Box Plot Chart	Neural Networks
Correlation Matrix	Parallel Latent Dirichlet Allocation